Karlsruhe Institute of Technology

Advanced Data Structures

Lecture 01: Bit Vectors

Florian Kurpicz

PINGO

https://pingo.scc.kit.edu/424928

Bit Vectors

Succinct Data Structures

- represent data structures space efficient
- close to their information theoretical minimum
- using every bit becomes necessary

Succinct Trees

- represent a tree with n nodes using only $2n$ bits
- navigation is possible with additional $o(n)$ bits

- storing a bit vector in practice is tricky
- `11011101` should require only a single byte

Efficient Bit Vectors in Practice (1/3)

`std::vector<char/int/...>`

- easy access
- very big: $1, 4, \ldots$ bytes per bit

Efficient Bit Vectors in Practice (1/3)

`std::vector<char/int/...>`

- easy access
- very big: $1, 4, \ldots$ bytes per bit

`std::vector<bool>`

- bit vector in C++ (1 bit per byte)
- easy access
- layout depending on implementation

Efficient Bit Vectors in Practice (1/3)

`std::vector<char/int/...>`

- easy access
- very big: $1, 4, \ldots$ bytes per bit

`std::vector<bool>`

- bit vector in C++ (1 bit per byte)
- easy access
- layout depending on implementation

`std::vector<uint64_t>`

- requires 8 bytes per bit(?)
- store 64 bits in 8 bytes
- how to access bits

Efficient Bit Vectors in Practice (1/3)

`std::vector<char/int/...>`

- easy access
- very big: $1, 4, \ldots$ bytes per bit

`std::vector<bool>`

- bit vector in C++ (1 bit per byte)
- easy access
- layout depending on implementation

`std::vector<uint64_t>`

- requires 8 bytes per bit(?)
- store 64 bits in 8 bytes
- how to access bits

- $i/64$ is position of 64-bit word
- $i\%64$ is position in 64-bit word

Efficient Bit Vectors in Practice (1/3)

`std::vector<char/int/...>`

- easy access
- very big: $1, 4, \ldots$ bytes per bit

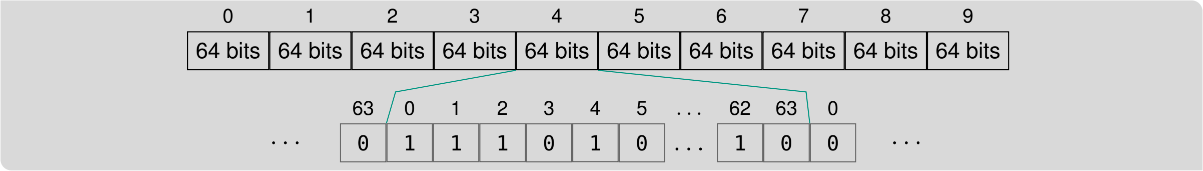
`std::vector<bool>`

- bit vector in C++ (1 bit per byte)
- easy access
- layout depending on implementation

`std::vector<uint64_t>`

- requires 8 bytes per bit(?)
- store 64 bits in 8 bytes
- how to access bits

- $i/64$ is position of 64-bit word
- $i\%64$ is position in 64-bit word


0 1 2 3 4 5 6 7 8 9
64 bits 64 bits 64 bits 64 bits 64 bits 64 bits 64 bits 64 bits 64 bits 64 bits
63 0 1 2 3 4 5 ... 62 63 0
... 0 1 1 1 0 1 0 ... 1 0 0 ...

Efficient Bit Vectors in Practice (2/3)

```cpp
// There is a bit vector
std::vector<uint64_t> bit_vector;

// access i-th bit
uint64_t block = bit_vector[i/64];
bool bit = (block >> ( 63 - (i % 64)) )  & 1ULL;
```

Efficient Bit Vectors in Practice (2/3)

```
// There is a bit vector
std::vector<uint64_t> bit_vector;

// access i-th bit
uint64_t block = bit_vector[i/64];
bool bit = (block >> ( 63 - (i % 64)) )  & 1ULL;
```

shift bits right

0	1	2	3	4	5	...	62	63
1	1	1	0	1	0	...	1	0

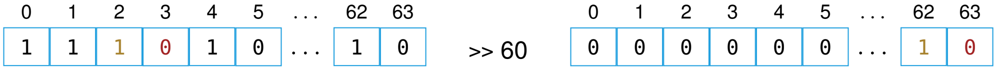
Efficient Bit Vectors in Practice (2/3)

```
// There is a bit vector
std::vector<uint64_t> bit_vector;

// access i-th bit
uint64_t block = bit_vector[i/64];
bool bit = (block >> ( 63 - (i % 64)) )  & 1ULL;
```

shift bits right

bits



0	1	2	3	4	5	...	62	63
1	1	1	0	1	0	...	1	0

>> 60

0	1	2	3	4	5	...	62	63
0	0	0	0	0	0	...	1	0

Efficient Bit Vectors in Practice (2/3)

```
// There is a bit vector
std::vector<uint64_t> bit_vector;

// access i-th bit
uint64_t block = bit_vector[i/64];
bool bit = (block >> ( 63 - (i % 64)) )  & 1ULL;
```

shift bits right

# bits

logical and 1

0	1	2	3	4	5	...	62	63
1	1	1	0	1	0	...	1	0

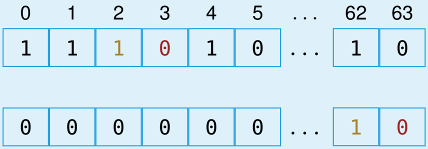
>> 60

0	1	2	3	4	5	...	62	63
0	0	0	0	0	0	...	1	0
							and	1

Efficient Bit Vectors in Practice (3/3)

`(block >> (63-(i%64))) & 1ULL;`

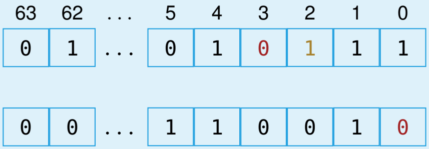
- fill bit vector from left to right



0	1	2	3	4	5	...	62	63
1	1	1	0	1	0	...	1	0
0	0	0	0	0	0	...	1	0

`(block >> (i%64)) & 1ULL;`

- fill blocks in bit vector right to left



63	62	...	5	4	3	2	1	0
0	1	...	0	1	0	1	1	1
0	0	...	1	1	0	0	1	0

Efficient Bit Vectors in Practice (3/3)

```
(block >> (63-(i%64))) & 1ULL;
```

- fill bit vector from left to right

0	1	2	3	4	5	...	62	63
1	1	1	0	1	0	...	1	0
0	0	0	0	0	0	...	1	0

```
(block >> (i%64)) & 1ULL;
```

- fill blocks in bit vector right to left

63	62	...	5	4	3	2	1	0
0	1	...	0	1	0	1	1	1
0	0	...	1	1	0	0	1	0

Efficient Bit Vectors in Practice (3/3)

```
(block >> (63-(i%64))) & 1ULL;
```

- fill bit vector from left to right

0	1	2	3	4	5	...	62	63
1	1	1	0	1	0	...	1	0
0	0	0	0	0	0	...	1	0

- assembler code:

```
mov  ecx,  edi
not  ecx
shr  rsi,  cl
mov  eax,  esi
and  eax,  1
```

```
(block >> (i%64)) & 1ULL;
```

- fill blocks in bit vector right to left

63	62	...	5	4	3	2	1	0
0	1	...	0	1	0	1	1	1
0	0	...	1	1	0	0	1	0

Efficient Bit Vectors in Practice (3/3)

```
(block >> (63-(i%64))) & 1ULL;
```

- fill bit vector from left to right

0	1	2	3	4	5	...	62	63
1	1	1	0	1	0	...	1	0
0	0	0	0	0	0	...	1	0

- assembler code:

```
mov  ecx,  edi
not  ecx
shr  rsi,  cl
mov  eax,  esi
and  eax,  1
```

```
(block >> (i%64)) & 1ULL;
```

- fill blocks in bit vector right to left

63	62	...	5	4	3	2	1	0
0	1	...	0	1	0	1	1	1
0	0	...	1	1	0	0	1	0

- assembler code:

```
mov  ecx,  edi
shr  rsi,  cl
mov  eax,  esi
and  eax,  1
```

Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of αs before i

$\text{select}_\alpha(j)$ position of j-th α

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank Queries on Bit Vectors (1/2)

$\mathrm{rank}_\alpha(i)$ # of αs before i

$\mathrm{select}_\alpha(j)$ position of j-th α

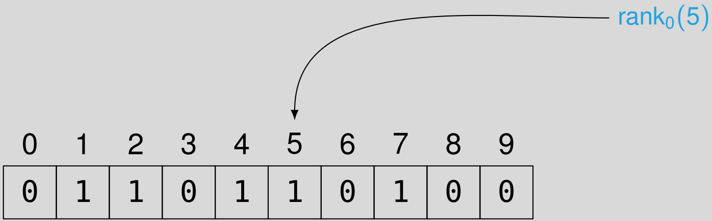
$\mathrm{rank}_0(5)$

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of αs before i

$\text{select}_\alpha(j)$ position of j-th α



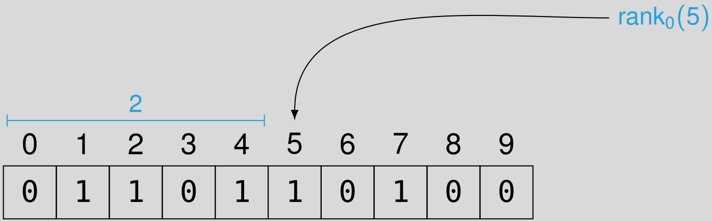
$\text{rank}_0(5)$

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of αs before i

$\text{select}_\alpha(j)$ position of j-th α

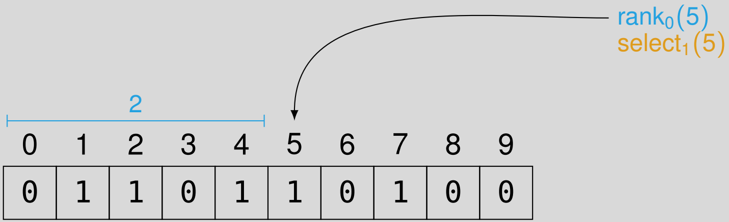


0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of αs before i

$\text{select}_\alpha(j)$ position of j-th α



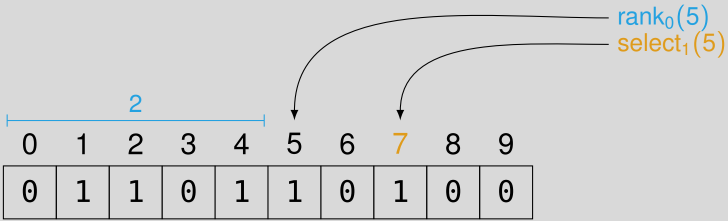
$\text{rank}_0(5)$
$\text{select}_1(5)$

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of αs before i

$\text{select}_\alpha(j)$ position of j-th α



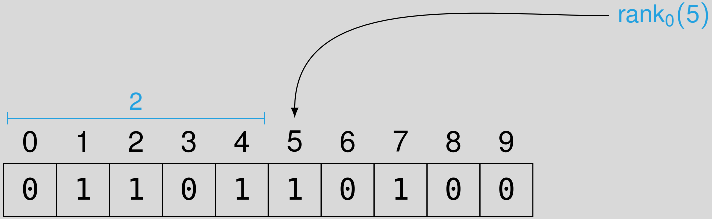
0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank Queries on Bit Vectors (1/2)

$\mathrm{rank}_\alpha(i)$ # of αs before i

$\mathrm{select}_\alpha(j)$ position of j-th α

PINGO-Frage

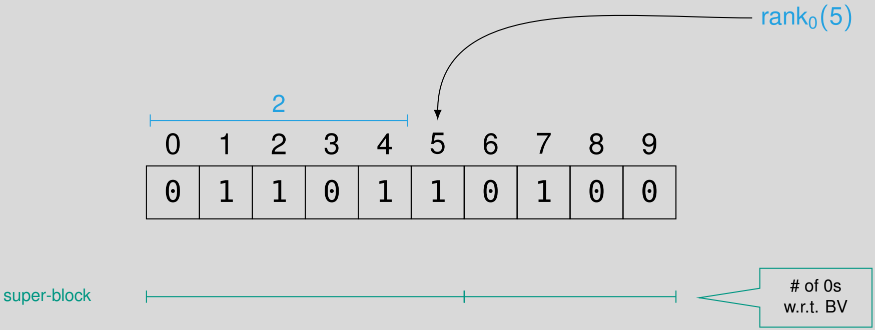


0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank Queries on Bit Vectors (1/2)

$\mathrm{rank}_\alpha(i)$ # of αs before i

$\mathrm{select}_\alpha(j)$ position of j-th α

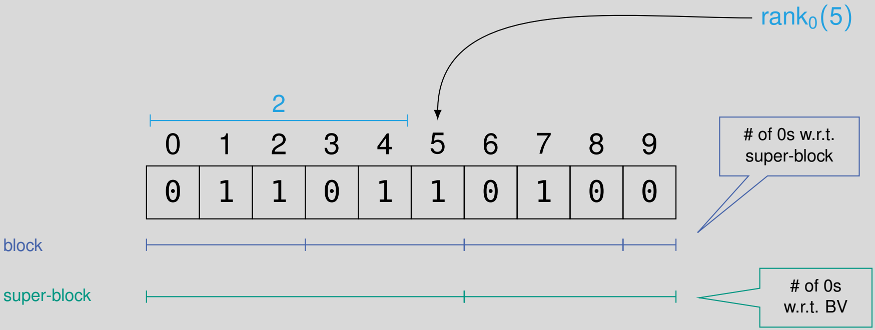


0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of αs before i

$\text{select}_\alpha(j)$ position of j-th α



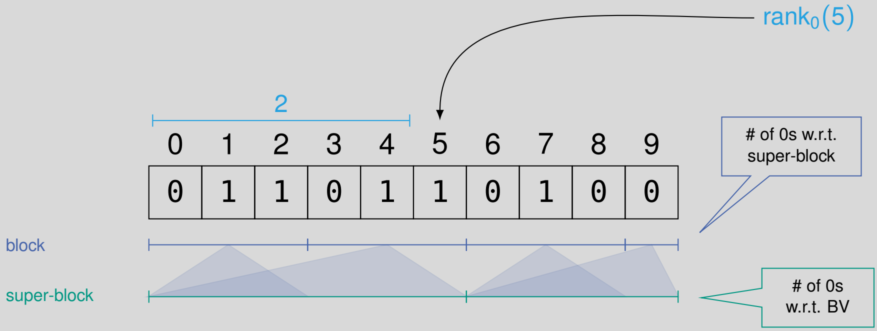
# of 0s w.r.t. super-block

of 0s w.r.t. BV

Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of αs before i

$\text{select}_\alpha(j)$ position of j-th α



# of 0s w.r.t. super-block

of 0s w.r.t. BV

Rank Queries on Bit Vectors (2/2)

- for a bit vector of size n
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$

Rank Queries on Bit Vectors (2/2)

- for a bit vector of size n
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$

- for all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ bits of space

Rank Queries on Bit Vectors (2/2)

- for a bit vector of size n
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$

- for all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ bits of space

- for all $\lfloor \frac{n}{s} \rfloor$ blocks, store number of 0s from beginning of super block to end of block
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ bits of space

Rank Queries on Bit Vectors (2/2)

- for a bit vector of size n
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$

- for all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ bits of space

- for all $\lfloor \frac{n}{s} \rfloor$ blocks, store number of 0s from beginning of super block to end of block
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ bits of space

- for all length-s bit vectors, for every position i store number of 0s up to i
- $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$ bits of space

Rank Queries on Bit Vectors (2/2)

- for a bit vector of size n
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$

- for all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ bits of space

- for all $\lfloor \frac{n}{s} \rfloor$ blocks, store number of 0s from beginning of super block to end of block
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ bits of space

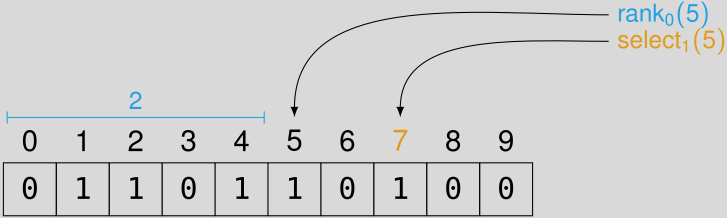
- for all length-s bit vectors, for every position i store number of 0s up to i
- $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$ bits of space

- query in $O(1)$ time
- $rank_0(i) = i - rank_1(i)$

Rank Queries on Bit Vectors (1/2)

$\text{rank}_\alpha(i)$ # of αs before i

$\text{select}_\alpha(j)$ position of j-th α



Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size n that contains k zeros
- **PINGO-Frage**

Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size n that contains k zeros
- **PINGO-Frage**
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $select_0(i) = S[i]$ ℹ if $k \in O(n/lgn)$ this suffice

Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size n that contains k zeros
- **PINGO-Frage**
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $select_0(i) = S[i]$ ℹ if $k \in O(n/lgn)$ this suffice

- better: k/b variable-sized super-blocks B_i, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) =$ $\sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b))$

Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size n that contains k zeros
- **PINGO-Frage**
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $select_0(i) = S[i]$ ℹ if $k \in O(n/lgn)$ this suffice

- better: k/b variable-sized super-blocks B_i, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b))$

- storing all possible results for the (prefix) sum
- $O((k \lg n)/b) = o(n)$ bits of space

Select in $o(n)$ Space and $O(1)$ Time

- select_0 in a bit vector of size n that contains k zeros
- **PINGO-Frage**
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $\mathit{select}_0(i) = S[i]$ ℹ if $k \in O(n/lgn)$ this suffice

- better: k/b variable-sized super-blocks B_i, such that super-block contains $b = \lg^2 n$ zeros
- $\mathit{select}_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + \mathit{select}_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b))$

- storing all possible results for the (prefix) sum
- $O((k \lg n)/b) = o(n)$ bits of space

- select on block depends on size of block
- $|B_{\lfloor i/b \rfloor}| \geq \lg^4 n$: store answers naively
  - requires $O(b \lg n) = O(\lg^3 n)$ bits of space
  - there are at most $O(n/\lg^4 n)$ such blocks
  - total $O(n/\lg n) = o(n)$ bits of space

Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size n that contains k zeros
- **PINGO-Frage**
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $select_0(i) = S[i]$ ℹ if $k \in O(n/lgn)$ this suffice

- better: k/b variable-sized super-blocks B_i, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor -1} |B_j| + select_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b))$

- storing all possible results for the (prefix) sum
- $O((k \lg n)/b) = o(n)$ bits of space

- select on block depends on size of block
- $|B_{\lfloor i/b \rfloor}| \geq \lg^4 n$: store answers naively
  - requires $O(b \lg n) = O(\lg^3 n)$ bits of space
  - there are at most $O(n/\lg^4 n)$ such blocks
  - total $O(n/\lg n) = o(n)$ bits of space
- $|B_{\lfloor i/b \rfloor}| < \lg^4 n$: divide super-block into blocks
  - same idea: variable-sized blocks containing $b' = \sqrt{\lg n}$ zeros
  - (prefix) sum $O((k \lg \lg n)/b') = o(n)$ bits
  - if size $\geq \lg n$ store all answers
  - if size $< \lg n$ store lookup table

Rank- and Select-Queries on Bit Vectors

Lemma: Binary Rank- and Select-Queries

Given a bit vector of size n, there exist data structures that can be computed in time $O(n)$ of size $o(n)$ bits that can answer rank and select queries on the bit vector in $O(1)$ time

Conclusion and Outlook

This Lecture

- bit vectors
- rank and select on bit vectors

Advanced Data Structures

BV

Conclusion and Outlook

This Lecture

- bit vectors
- rank and select on bit vectors

- efficient bit vectors in practice

Advanced Data Structures

BV

Conclusion and Outlook

This Lecture

- bit vectors
- rank and select on bit vectors

- efficient bit vectors in practice

Next Lecture

- succinct trees using bit vectors
- navigation in succinct trees

Advanced Data Structures

BV